Mining Interprocedural, Data-Oriented Usage Patterns in JavaScript Web Applications

Hung Viet Nguyen hungnv@iastate.edu
Hoan Anh Nguyen hoan@iastate.edu
Anh Tuan Nguyen anhnt@iastate.edu
Tien N. Nguyen tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA

ABSTRACT
A frequently occurring usage of program elements in a programming language and software libraries is called a usage pattern. In JavaScript (JS) Web applications, JS usage patterns in their source code have special characteristics that pose challenges in pattern mining. They involve nested data objects with no corresponding names or types. JS functions can be also used as data objects. JS usages are often cross-language, inter-procedural, and involve control and data flow dependencies among JS program entities and data objects whose data types are revealed only at run time due to dynamic typing in JS. This paper presents JSModel, a novel graph-based representation for JS usages, and JSMiner, a scalable approach to mine inter-procedural, data-oriented JS usage patterns. Our empirical evaluation on several Web programs shows that JSMiner efficiently detects more JS patterns with higher accuracy than a state-of-the-art approach. We conducted experiments to show JSModel’s usefulness in two applications: detecting anti-patterns (buggy patterns) and documenting JS APIs via pattern skeletons. Our controlled experiment shows that the mined patterns are useful as JS documentation and code templates.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

General Terms
Algorithms, Design, Documentation, Languages

Keywords
Usage Patterns, Mining, JavaScript, Web Applications

1. INTRODUCTION
When writing code, developers often use the program elements in a programming language and the APIs of software libraries in a specific order to realize a certain task. For example, in a task of writing a number to a data file in C, a variable of type FILE must be opened (via the fopen function) before another variable of type int can be written to the file (via the fwrite function), and finally the file should be closed to conclude the task (via the fclose function). Multiple programs could share a common task, and software developers repeat the code to realize it, following the usage of the program elements for that task. A frequently occurring code for a common task is called a usage pattern.

Several methods for mining usage patterns from code repositories have been proposed. The resulting usage patterns have been shown to be useful in many software engineering tasks, e.g., usage documentation [2], code smell and anomaly detection [7], defect and vulnerability detection [4, 5, 6, 11, 24], code search [16], code completion [28], evolution adaptation [3], etc. However, existing usage pattern mining methods are limited in the context of JavaScript (JS) Web code. The usage patterns in JS have characteristics that pose several challenges in pattern mining. First, JS usage patterns involve control and data flow dependencies among program entities and data objects whose data types are often revealed only at run time due to dynamic typing in JS. This creates great challenges since existing mining algorithms require named types for matching the entities in the usages [2]. Second, JS usage patterns can involve aggregated and nested data objects, which are often inline code, i.e., having no corresponding named entities in the program. Thus, a mining algorithm must match the structures/prototypes of objects, and cannot rely solely on the type names. Despite recent advance in data clone detection for spreadsheets [30], data objects have never been considered in existing pattern mining approaches for the traditional typed languages since they can be abstracted with their types. Third, a JS pattern may cross the boundary of functions due to a popular programming style in JS for event handlers. Several static analysis-based mining techniques could not handle such inter-procedural patterns due to their scalability. Additional complexity is from the fact that JS functions can be used as data objects. Finally, JS code is used to provide the control logic for Web pages, thus, JS patterns often involve HTML elements. No existing mining approach can handle all those properties in JS patterns yet.

This paper presents JSMiner, a static approach to mine inter-procedural, data-oriented JS usage patterns in Web applications. A JS usage is modeled by a novel graph-based representation, called JSModel. In addition to function calls, field accesses, and control nodes as in existing usage models [2, 7],
JSModel has (un)named data nodes for variables, object literals, JS functions, and HTML elements. The structure of an object is captured via (un)named data nodes and edges that represent their containment relations. Edges are also used to model control and data flow dependencies among nodes. A data node for a JS function, which has an edge to its body, can be connected as an input of any operation, e.g., as an argument of a function call. This facilitates the mining of inter-procedural patterns and the patterns with anonymous functions in event handlers. Our graph-based mining tool, JSMiner, is tailored toward JS to be efficient and scalable for large usage graphs. It relies on graph structures and labels for matching, rather than data types as in existing tools.

Our empirical evaluation on several Web programs shows that JSMiner can detect more JS patterns with higher accuracy and efficiency than the state-of-the-art tool GrouMiner [2]. Other experiments also showed JSModel’s usefulness in two applications: detecting buggy patterns and documenting JS APIs via patterns. Our key contributions include:

1. JSModel, a graph-based representation model for JS usages and patterns in JS Web applications (Sections 3-4).
2. JSMiner, a scalable JS pattern mining tool (Section 5).
3. An empirical evaluation on its scalability and accuracy in pattern mining in Web programs, and JSModel’s usefulness in anti-pattern detection and JS documentation (Section 6).

2. MOTIVATING EXAMPLES

Let us present a few usage scenarios that share JS usage patterns, and then discuss the challenges in mining them.

2.1 Usage Scenario 1

Figure 1 shows the HTML/JS code of an example from the website of the Google Maps JS API library [1]. The library allows developers to embed Google Maps applications in their websites by providing a variety of utilities for manipulating maps and adding content to the maps. This example Web page displays a map at a given location on the left panel. When a user selects two points A and B on the map, the optimal driving directions from A to B will be calculated and highlighted, and the direction details will be displayed on the right panel. To provide this capability, the Google Maps API was used.

**Step 1. Creating two HTML panels for the content of the map and the direction information:** In Figure 1, the HTML `<div>` tags `map_canvas` and `directionsPanel` on lines 36-37 serve as the place holders for the map and direction panels.

**Step 2. Creating a JS Map object (provided by the API) and displaying it on the HTML panel for the map:** This step is realized in lines 6-13 of the function `initialize`, which will be called to create the actual map as the web page is loaded (line 35). The variable `myOptions` is first initialized with an object literal storing the properties of the map (lines 7-12). (A JS object literal is an `unnamed` list of property-value pairs of an object, enclosed in curly braces.) Then, the variable `myOptions` is passed as the second argument to a function that creates a Map object (line 13). To display the map on the map panel, `document.getElementById(“map_canvas”)` is passed as the first argument of that function (line 13).

**Step 3. Creating a JS DirectionsRenderer API object to highlight the route on the map and displaying the direction on the HTML direction panel (lines 5, 14-15):** The variable `directionsDisplay` is first initialized as a DirectionsRenderer object (line 5), which contains direction details between two points. Functions `setMap` and `setPanel` are called to display the direction on the map and direction panels (lines 14-15).

**Step 4. Updating the directions when the user selects or updates two locations:** The API function `addListener` (lines 16-18) is called to declare a handler for the event when a user clicks to select two points on the map.

2.2 Usage Scenario 2

We also found a Google Maps API example on [1] that allows users to mark an area on a map with a polygon (Figure 2). The web page example has different functionality but shares the `map displaying task` with the previous example. Following the Google Maps API usage, the authors of that web page example realized that task with the same four steps. Step 1 was implemented on lines 18-19, step 2 on lines 4-9, step 3 on line 11, and step 4 on line 13.

**Observations.** From the two above usage scenarios, we have the following observations:

- **O1.** A library provides usages for different tasks in different scenarios. A usage for one task can be shared among multiple scenarios. The frequently occurring usage for a common task is called a usage pattern. As writing code in a usage, developers need to follow specific steps and use program elements in specific orders. That is, there are control flow dependencies among the actions in those usages. For example, the Map object needs to be created before it can be attached to another object via the function `setMap`. There...
are also **data flow dependencies** among actions, e.g., the data resulted from an action can be the input of another. Moreover, the code for a pattern is not necessarily contiguous since it can be intermixed with the code for other tasks.

**O2.** A JS usage pattern often involves not only **function calls** but also **JS data objects and JS functions** that are required for the intended usage. For instance, a JS object literal is needed to initialize the properties of the Map object, and a JS event handler (whether anonymous or named) is used as an argument of `addListener`. Multiple objects can be aggregated as the values of the properties of an *unnamed object literal*. Note that *JS functions are treated as data* and can be passed as arguments to other functions.

**O3.** A JS usage pattern can involve code written in different languages, and there are **cross-language dependencies** among those code fragments. For instance, the HTML element `<body onload="initialize()" ... >` specifies a call to the JS function `initialize` on its `onload` event, whereas the JS code `document.getElementById('map_canvas').appendChild(new google.maps.Map(document.getElementById('map')));` accesses an HTML element with the id `map_canvas`.

**O4.** The same usage pattern might have **implementation variants** (written in slightly different ways). For example, in Figure 1, the object literal describing the map’s properties is assigned to variable `myOptions`, and `myOptions` is passed as an argument to `google.maps.Map` (lines 7-13), while at lines 5-9 of Figure 2, the object literal is directly passed as an argument. The event handler at line 16 of Figure 1 is anonymous, while the one at line 13 of Figure 2 is a named function.

**O5.** JS is a **dynamically typed** language. The data types of JS variables can be determined/changed automatically as needed during the execution. For instance, the variables `poly` and `map` (line 2, Figure 2) are declared with JS keyword `var`, without their types being specified in the code. At run time, `map` will be initialized as a `Map` object (line 5, Figure 2).

### 2.3 Usage Scenario 3

Figure 3 displays another Google Maps example, which shares the same route calculation/displaying task as in the first usage scenario except for a few details, e.g., the source and destination locations are now pre-defined. The usage for that task is inside the function `calcRoute` (lines 2-15, Figure 3), similar to the one on lines 20-33, Figure 1.

**O6.** A usage pattern can involve **inter-procedural dependencies**. For example, the route calculation usage pattern involves both the code from outside an event handler (lines 3-10) and its function body (lines 11-12). This type of inter-procedural pattern occurs in JS **event handling** code, which is popular in Web applications to enhance interactivity. Thus, all or parts of such a JS callback function belong to a usage.

### 2.4 Challenges in Mining JS Usage Patterns

These examples illustrate new challenges in mining usage patterns in JS-based Web code. First, a pattern may contain (aggregated) data objects including unnamed object literals (O2). Worse, some JS variables might not have declared types (O5). Thus, pattern mining that relies solely on matching type names for program entities would not work. The prototypes of data objects must be considered during entity matching to find patterns. Data objects, which are crucial parts of JS usage patterns, have never been considered in existing mining approaches for the traditional typed languages since they can be abstracted with their types. Second, a JS pattern often contains inter-procedural dependencies due to event handling code (O6). Anonymous functions are used both as functions and as data when passed as arguments to another function. Thus, mining inter-procedural patterns is needed in JS code. However, this not only creates the scalability issue, but also increases the complexity in a mining tool to represent such complex relationships. Moreover, in some applications, anonymous functions/objects are used, and in others, the named ones are (O4). A mining tool must be able to detect patterns despite that difference. Finally, a JS pattern involves also non-JS web code (O3).

### 3. REPRESENTATION OF JS USAGES

To address those challenges, we present JSModel, a graph representation, and JSMiner, a pattern miner for JS usage.

**Definition 1 (JSModel).** A **JSModel** is a labeled, directed graph representing a usage in a JS program. In a JSModel, the nodes represent program actions, data, and control structures; and the edges represent control and data flows, and inclusion relationships among them.

All types of JSModel nodes and edges are listed in Table 1. There are 3 types of nodes: action, data, and control nodes.

1. **Action nodes** (N1-N3). Program actions in a JS usage include function invocations, variable assignments, and
There are three edge types (control, data, and inclusion):

1. **Control-flow edges (E1).** Control-flow edges represent the control flows (orders) among the actions and control structures in a usage. Such edges have no labels. For example, in Figure 1, variable `request` is initialized (line 21), then is passed to function `route` (line 27). In Figure 4, the order is modeled as a control-flow edge from "=" to `route`.

2. **Data-flow edges (E2).** Data-flow edges with their labels model different types of data flow dependencies between the actions and data entities in a usage, e.g., 1) an arg edge from a variable `x` to a function call node `m` indicates that `x` is an argument of `m`; or 2) an obj edge between `m` and `x` means that `m` is a function call on the object `x`. For instance, the data-flow edge labeled `out` from the action node "=" to the data node `request` in Figure 4 means that a value is assigned to variable `request`.

3. **Inclusion edges (E3).** An inclusion edge (labeled as incl) from a node `x` to a node `y` indicates that `x` has a block of code directly containing `y`, where `x` can be a function, object literal, or control structure. In Figure 4, the control node `IF` has two inclusion edges to the two actions `clearMarkers` and `setDirections` to indicate that those two function calls need to be placed inside an if statement (lines 28-31, Figure 1). A function node also has incl edges to its body’s statements.

Let us explain in detail different node and edge types in a JSMModel and our design rationale, and illustrate how those design ideas address the challenges discussed in Section 2.

### 3.1 Representing Data Objects

We introduce three data node types (namely Variable, Function, and Object Literal) to represent those JS data objects, and HTML nodes to represent HTML elements.

**Variable nodes (N4, Table 1).** A Variable node is used to model a variable or a field of an object. If the data is an object field, the corresponding JSMModel will have a data node representing the object, a data node representing the object’s field, and a data-flow edge between them.

**Object literal nodes (N6).** In JS usages, object literals are unnamed. They must be matched via their prototypes (structures). Thus, JSMModel uses an ObjLit node. Each object literal has one or multiple fields, each of which has a name and an initial value. Thus, an ObjLit node has inclusion edges referring to the object’s fields. The value initialization of each field is modeled by the "=" action nodes. For example, the ObjLit node in Figure 4 represents the object literal at line 21 (Figure 1). The field of an object literal can in turn be assigned with another object literal, i.e., object literals can be nested. JSMModel can model a usage with nested object literals, as in the following example.

```javascript
circle = {
   radius : getRadius () ,
   center : {
      x : 1 ,
      y: 2
   } 
}
```

Besides the above data types, a JS usage may also contain primitive types. Since data values might vary in different usages of the same pattern, JSMModel does not represent them.

**HTML nodes (N7).** We also represent HTML elements as data nodes of type HTML. Information about an element’s properties (e.g., its type, name, and ID) is recorded, so that when it is accessed from JS code, the corresponding HTML node can be used to represent the referred element.

---

Table 1: Types of nodes and edges in a JSMModel

<table>
<thead>
<tr>
<th>Type</th>
<th>Sub-Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODES</td>
<td>N1. Action</td>
<td>FuncCall (m) m is a function invocation</td>
</tr>
<tr>
<td></td>
<td>N2. Action</td>
<td>Assignment (=) A value assignment to a variable</td>
</tr>
<tr>
<td></td>
<td>N3. Action</td>
<td>Operation (OP) A data operation (+, !=, ...)</td>
</tr>
<tr>
<td></td>
<td>N4. Data</td>
<td>Variable (x) x is a variable or a field of an object</td>
</tr>
<tr>
<td></td>
<td>N5. Data</td>
<td>Function A function (used as data)</td>
</tr>
<tr>
<td></td>
<td>N6. Data</td>
<td>ObjLit An object literal</td>
</tr>
<tr>
<td></td>
<td>N7. Data</td>
<td>HTML An HTML tag element</td>
</tr>
<tr>
<td></td>
<td>N8. Control</td>
<td>A control structure (if, for, while)</td>
</tr>
</tbody>
</table>

---

EDGES (x → y)

| E1. Control | Control-flow dependency from x to y |
| E2. Data    | Data-flow dependency from x to y |
| E3. Inclusion | x has a block of code including y |

---

**Figure 4:** JSMModel for the body of `calcRoute` in Figure 1

data operations (e.g., ‘+’, ‘==’, ‘!=’, etc). Those actions are represented by action nodes in a JSMModel. The label of an action node is either "=" for an assignment, the function name for a function invocation, or the notation OP for a data operation. For example, the action nodes at L21 and L27 in the JSMModel in Figure 4 correspond to the assignment and the function invocation route on lines 21 and 27 in Figure 1.

2. **Data nodes (N4-N7).** Data nodes are used to represent the data involved in the actions of a JS usage. JSMModel has four types of data nodes: Variable, Function, and Object Literal for representing JS data objects, and HTML data nodes for HTML elements. The label of a data node contains the name/ID of the corresponding data entity (i.e., variable, function, object literal, or HTML element) if it has a name/ID. Otherwise, a label “anonymous” is used.

3. **Control nodes (N8).** Control nodes model control structures such as branches and loops that are required for the intended usage. The label of a control node is the name of its corresponding control structure (e.g., IF or WHILE).
Function nodes (N5). We represent a function as a data node of type Function. To support inter-procedural usage pattern, a Function node is linked via inclusion edges to the JSModels describing its body code (the Function node in Figure 4 connects to the inside of the JSModel for its body). There are two ways that a function can be used. First, a function is invoked via a function call. Thus, an action node FuncCall (N1, Table 1) is connected via a data-flow edge to its corresponding Function node (N5), if the declaration of that function is found in the current source code, so that inter-procedural dependencies in a JS usage can be recognized. Second, if a JS function is used as a data object, the Function node will connect to other nodes. For example, in Figure 4, the Function node connects via a data-flow edge to the action node route since the function is an argument of route. A function could also have a name and be referred to, as in the following code from Figure 2, where the function addPoint is used as an argument of addListener:

```javascript
function addPoint(event) {
    // JS code
    // HTML code
    // JS code
    // HTML code
}
```

In such cases, the declared function and its references are represented by a single Function node as in the model below.

![Diagram](image)

Function nodes and incl edges are needed in JSModel because in many common cases in JS code where an event handling function is used (anonymously or not), a usage pattern involves (part of) the function’s body (see Section 2.3). Even if the handler is named, using only its name is not enough since a different usage of the same pattern might use different names which are assigned to the same body. This is possible since a function is treated as a data entity. Function types cannot be used either due to JS dynamic typing.

### 3.2 Interprocedural & Inclusion Dependencies

As explained on the Function node, the inclusion edges support inter-procedural usage patterns. Moreover, inclusion edges are also defined for all actions, data, and control structures that interact across the boundary of a block of code (enclosed by curly braces). For example, the ObjLit in Figure 4 has incl edges connecting it to the four data nodes origin, destination, waypoints, and travelMode. This allows JSModel to maintain the fields of an object literal. Such connections among nodes also allow a usage pattern to be recognized even though it involves actions and data that are inter-procedural or nested at multiple levels of blocks.

### 3.3 Implementation Variants & Dynamic Types

JS variables have dynamic types, and the same usage pattern may be written using different variable names. A function can be used anonymously or as a named entity. To address such implementation variants, JSMiner recognizes a usage pattern based on the values of the variables in the usages rather than their types or names, and based on the bodies of the functions, rather than their names. This helps us detect a pattern both when the values are assigned to variables and when they are inlined directly in code (e.g., as unnamed object literal), without being assigned to any variables.

To achieve that, JSMiner analyzes the program’s statements in their order of execution and records the propagation of variables’ values through assignments. When there is a data flow dependency from a variable to an action or another data entity, we use the variable’s value for that dependency. For example, in Figure 1, the variable request is assigned with an object literal (line 21), and is passed to the function call route (line 27). Thus, that object literal is the value of request and is used as an argument of the function call route. To model that, in Figure 4, an arg edge connects the ObjLit node (representing the value of request) to route.

When building a JSModel, the variables’ names are needed to recognize data flow dependencies with the actions and other data entities in a JS usage. However, different usages of the same pattern may use different variable names. Thus, during the pattern mining step, the labels of the respective Variable nodes are anonymized so that usages are compared by their graph structures rather than the variables’ names. For instance, the node request in Figure 4 will be anonymized.

If the value of a variable or the declaration of a function cannot be resolved, JSMiner keeps its name in JSModel. The rationale is that such a variable/function is usually from a library and its name remains the same for all usages of the same pattern. Thus, it is sufficient to match them by names.

### 3.4 Cross-language Dependencies

There are two types of cross-language dependencies between JS and HTML entities. First, an HTML element is defined with an ID and referred to from JS code via an HTML DOM access. For example, in Figure 1, an HTML \(<div>\) with ID= "map_canvas" is defined at line 36, and accessed by document.getElementById("map_canvas") at line 13. In such cases, HTML data nodes are created and connected to the corresponding action nodes as in the JSModel below:

![Diagram](image)

Second, the dependency between HTML and JS code also occurs when JS code is embedded in an HTML event handler. The code below from Figure 1 illustrates this case.

```javascript
function initialize () {
    // JS code
    <body onload="initialize ()"> // HTML code
}
```

As seen, the handler for the onload event of the HTML \(<body>\) tag has a call to the JS function initialize, which is declared in JS code. To model this dependency, we leverage the fact that HTML elements and their properties can be created/modified dynamically from JS code. Thus, the above HTML/JS code is equivalent to the following JS code:

```javascript
function initialize () {
    // JS code
    document.body.onload = function() { initialize () }; // JS code
}
```

Since the two ways of assigning an event handler to the \(<body>\) tag are interchangeable, we represent it as follows:
4. BUILDING JSModel

The JSModel for a JS code portion is composed of the JSModels for the smaller portions of that code, and the edges connecting those JSModels represent their dependenciea. A JSModel has a designated core node. Connecting two JSModels J₁ and J₂ means that the core node of J₁ is connected to that of J₂. To build a JSModel, we parse the given code using Eclipse’s JS parser. Table 2 shows the building rules for different AST structures. Core nodes are highlighted in bold. JS statements/expressions are processed as follows.

1. Variable. For a JS variable, we either create a new Variable node with the variable’s name as its label or reuse the Variable node if a variable was already created for a variable with the same name and in the same program scope.

2. Field access. For a field access E.F, we first check whether the expression E is a variable that has been initialized. If it is, JSMiner connects the JSModel of E, J(E), to a Variable node modeling the field F via an obj data-flow edge. Otherwise, JSMiner assumes that the field access is a labeled property of HTML data node and stores the tag’s name, type, and ID. For a function call or object creation (e.g., new Map()), JSMiner models the call’s condition via a data-flow edge and to the JSModel of the function’s body statements via inclusion edges.

3. Assignment. For a variable assignment, JSMiner creates a Assignment node linking to the JSModel of the object fields by inclusion edges. The initialization of a field F₁ with an expression E₁ is modeled by an Assignment node linking to J(E₁) and J(F₁).

4. Object literal. JSMiner creates an ObjectLit node and connects it to the JSModels of the object fields by inclusion edges. The initialization of a field F₁ with an expression E₁ is modeled by an Assignment node linking to J(E₁) and J(F₁).

5. Block. JSMiner connects the statements in their appearance order in a block to form a closure by control-flow edges. For nested code blocks, only the statements in the same level are connected. The core nodes of a block include those of the statements in that block.

6. Function declaration. A function data node is built with the function’s name as its label. It is connected to the JSModels of the parameters via data-flow edges and to those of the function’s body statements via inclusion edges.

7. Function call. A function call or object creation (e.g., new Map()) is modeled by a FuncCall action node with the function’s name as its label. Data-flow edges are used to connect the FuncCall node to the JSModels of the function’s arguments and that of the object whose function is invoked. Moreover, if the function’s declaration is found in the code, a FuncCall node will be linked to the respective Function node via an is-decl data-flow edge. JSMiner models special control statements, e.g., break and return, as special FuncCall nodes.

8. Pre-/Post-/Infix expression. This expression is modeled by an OP action node. The sub-expressions of this expression are linked to the OP node via data-flow edges.

9. While statement. A WHILE control node is created for this statement. It is connected to the JSModel of the while’s condition via a data-flow edge and to the JSModels of its body statements via inclusion edges. Other loop statements, e.g., do and for, are handled similarly.

10. If statement. A process similar to case 9 is applied. JSMiner further annotates the inclusion edges with either T and F to distinguish between two branches. Switch statements and conditional statements are handled similarly.

Building JSModel for embedded JS code. The process has three steps. First, JSMiner parses the HTML code, and for each HTML tag, it creates a corresponding HTML data node, and stores the tag’s name, type, and ID. Second, it identifies any JS code embedded within HTML <script> tags. JS files that are included by the src property of HTML <script> tags are not extracted since they are considered as libraries. For JS code embedded in an HTML event handler, e.g., onload and on click, it builds the JSModel as described in Section 3.4. Finally, it builds the JSModels for the extracted JS code portions. In the process, when JS code accesses an HTML element, the corresponding HTML node created in step 1 will be used in the JSModel for that JS code.

5. JS PATTERN MINING ALGORITHM

Given the JSModels built from various JS usages, our pattern mining algorithm, JSMiner, identifies usage patterns that frequently appear in their JS code. It is inspired from GroupMiner [2] with two key ideas. First, small patterns are extended to detect larger patterns. Second, the expansion is done based on the specific characteristics of JSModel nodes and edges to improve performance and detect meaningful JS patterns. Figure 5 shows the pseudo code of the algorithm. To handle implementation variants of the same usage pattern, JSMiner first removes the labels of the JSModel nodes of certain variables and functions (Section 3.4) so that the JSModels for the same usage pattern are isomorphic even if its instances use different variables or functions’ names (line 2). Then, it collects all patterns of size 1 (having one node) into the set S of discovered patterns (line 3). For each size-1 pattern p, it calls the recursive function ExtendPattern to extend p and discover patterns of larger sizes (line 4).

To identify patterns of size k + 1 from a pattern p of size k, JSMiner first generates candidate instances of size k + 1 from the instances of p (line 7). Then, it clusters these instances into groups of isomorphic graphs (the Cluster function in line 8), in which a group q represents a candidate pattern. Since detecting isomorphic subgraphs is NP-complete, we
6. EMPIRICAL EVALUATION

To evaluate JSMiner’s performance, we conducted empirical experiments on various subject codebases (column Code Base in Table 3). The first set of subjects is taken from real-world code examples of several JS libraries, which are available on the Web sites of those libraries or in their code repositories. For the second set, we ran Crawljax [25], a Web crawling tool, (with depth 2) to retrieve various pages from several popular Web sites. Columns Files and JS Funcs show the numbers of extracted HTML/JS files and JS functions.

6.1 JSMiner’s Accuracy Sensitivity Analysis

First, we aimed to evaluate JSMiner’s pattern detection accuracy in response to the threshold $\sigma$ of the occurrence frequencies of the usages. For each codebase, we varied $\sigma$ and ran it to detect patterns. We then manually examined the detected patterns and identified the (in)correct patterns according to the documentation and specifications of the corresponding JS libraries. Precision is measured by the ratio of the correctly detected patterns over all detected ones.

In Figures 6a-c-e, for the three selected codebases (Google Maps, Closure Library, and D3.js), as the frequency threshold $\sigma$ decreases, the number of correctly detected patterns tends to be higher, i.e. recall increases. As $\sigma$ increases, the number of incorrect patterns decreases, and precision tends to rise and reaches 100% when $\sigma$ is sufficiently large (Figures 6b-d-f). For example, in Google Maps, when $\sigma$ was at 6, JSMiner detected a total of 70 patterns with 70% precision, and when $\sigma$ was at 17, it detected 7 patterns with 100% precision. The running time ranges from a few seconds to a few minutes and generally decreases with higher thresholds (Figures 6b-d-f).

6.2 JSMiner’s Performance Evaluation

To evaluate the quality and complexity of the patterns detected by JSMiner, we repeated the above process for all other subject codebases. For each subject, we chose a frequency threshold $\sigma$ such that it produced a set of high-quality patterns (with at least 80% precision) while obtaining as many patterns as possible. Among all of those patterns, the sets of correctly detected patterns are reported in Table 3. Column $\sigma$ is for the frequency threshold, and $f_{max}$ is the highest frequency of the detected patterns. The total number of patterns (Total) includes those of different sizes with 5-8, 9-15, or more than 15 nodes (see the respective columns). To see the characteristics of the detected patterns, we counted those that contain inclusion edges (incl), functions being used as data (func), object literals (objlit), cross-language dependencies (cross), and control structures (ctrl).

As seen, JSMiner detected a total of 246 correct JS patterns with high complexity. The size of the patterns is typically 5-15 nodes, with 14 patterns of size larger than 15. The majority have inter-procedural dependencies (163/246 patterns) and use JS functions as data objects (132/246). Many patterns involve cross-language dependencies between JS and HTML (36), control structures (67), and object literals (15). Importantly, JSMiner is capable of handling codebases with over 200 files and 3,600 JS functions (columns Files and JS Funcs) and large graphs, some having more than 44,000 nodes and edges (column JSModel Size), and with efficient time (<50 seconds) and memory usage (<500MB). The complete results can be found on JSMiner’s Web site [29].

6.3 Comparison with GrouMiner

We conducted another experiment to compare JSMiner with GrouMiner [2], which is a general, graph-based, usage pattern mining algorithm. Its tool is for Java. We

---

Figure 5: JSModel usage pattern mining algorithm

use a vector-based approach, Exas [2], in which a subgraph has a characteristic vector computed from the occurrences of the sequences of its nodes and edges’ labels and types. Two graphs are considered isomorphic if they have the same vectors. If the frequency of a candidate pattern $q$ is larger than a threshold $\sigma$ and $q$ is not a discovered pattern, $q$ is identified as a new pattern and used to mine larger ones (lines 9-11).

When extending a pattern $p$ to identify larger patterns, JSMiner generates candidate pattern instances from $p$ (lines 13-18). To do this, it extends each instance of $p$ by adding to it the nodes returned by the NextNodes function (line 16).

A new instance $c$ generated from an instance $i$ and a node $n$ (denoted as $i \oplus n$) is a graph containing $n$, all the nodes and edges in $i$, and all edges connecting $n$ to $i$ (line 17). The NextNodes function (returning a list of nodes, lines 19-29) is used to prioritize the list $L$ of the nodes that will be added to an instance $i$ to generate new candidate pattern instances. Since we traverse in a depth-first search order, the prioritization aims to detect meaningful JS patterns, and to help reduce the detection time. Specifically, for a given node $x$ in the current instance, it determines the key edges that should be included first in the extended instance based on the node type of $x$. Then, the key nodes that are connected to $x$ via those key edges are set with top priorities (lines 22-27). For example, for an action node of type FuncCall, JSMiner prioritizes its data-flow edge is-decl, which points to a Function node containing the body of the invoked function (line 23), attempting to detect an inter-procedural pattern. Similarly, for an ObjLit, it uses incl edges to set high priorities to the nodes for the object literal’s fields (line 25). Finally, the other neighboring nodes of the instance $i$ are added to $L$ with lower priorities than the key nodes (line 28).
thus re-implemented its algorithm for JS. We followed the same process as in the sensitivity analysis. Figure 7 shows their comparison results on Google Maps. As the frequency threshold $\sigma$ increases, the precision of both GrouMiner and JSMiner increases (Figure 7b). However, at any given $\sigma$, JSMiner detected more correct patterns (Figure 7a) with higher precision (Figure 7b). Examining the result, we found that GrouMiner could not detect JS patterns having cross-language and inter-procedural dependencies, and functions being used as data. Variables without declared types caused it to incorrectly detect patterns. Since JSMiner built larger graphs, its running time was almost double that of GrouMiner.

6.4 Case Studies
Let us present some JS usage patterns detected by JSMiner.

1. “Sending an AJAX request” pattern in MooTools (Figure 8). Figure 8a shows a JS usage for sending an AJAX request to a server (lines 4-6) as a user clicks on a loaded Web page. The code is wrapped by an anonymous function and attached to the click event of an HTML element via function $\text{addEvent}$ (line 2). $\text{addEvent}$ itself is wrapped by another anonymous handler (line 1). As seen, the detected pattern (Figure 8b) involves inter-procedural dependencies at multiple levels (with two nested event handlers). The event handlers are passed as data to other functions ($\text{addEvent}$ on lines 1 and 2). Several instances of this pattern occurred for other types of the objects that send the AJAX requests.

2. “Logging” in Closure Library (Figure 9). This usage contains a pattern (highlighted) that outputs message logs on an event. First, a $\text{Logger}$ and event types are defined (lines 1, 4-5). Then, a function to output the logs is declared (lines 6-8). Finally, that function is attached to the event of an HTML element (line 11). As seen, the pattern involves the interaction of multiple objects ($\text{logEvent}$, $\text{EVENTS}$, and $\text{gEvent}$). The declaration of the function $\text{logEvent}$ (including its body) (lines 6-8) and its use as a data object (line 11) are parts of the pattern. JSMiner detected this inter-procedural pattern even though it is intermixed with other code.

3. “Resetting zoom level” in jqPlot (Figure 10). This pattern (highlighted) demonstrates cross-language interaction between HTML and JS code. In the JS code, a $\text{plot}$ variable is initialized (line 5) on the event that the HTML document is ready (line 2). In the HTML code, as a button is clicked, $\text{resetZoom}$ will be invoked (line 8). That function embedded in HTML takes $\text{plot}$ defined in JS as its argument.

4. “Playing video” in www.apple.com (Figure 11). Figure 11 shows two usages of the pattern to play a video.
<table>
<thead>
<tr>
<th>Code Base</th>
<th>Files</th>
<th>Func</th>
<th>avg</th>
<th>max</th>
<th>σ</th>
<th>f_max</th>
<th>Total</th>
<th>5-8</th>
<th>9-15</th>
<th>&gt;15</th>
<th>incl</th>
<th>func</th>
<th>objet</th>
<th>cross</th>
<th>ctrl</th>
<th>(MB)</th>
<th>(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure Library</td>
<td>117</td>
<td>594</td>
<td>327</td>
<td>4,744</td>
<td>10</td>
<td>26</td>
<td>14</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>428</td>
</tr>
<tr>
<td>D3.js</td>
<td>168</td>
<td>1,126</td>
<td>335</td>
<td>2,876</td>
<td>25</td>
<td>38</td>
<td>29</td>
<td>27</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>283</td>
</tr>
<tr>
<td>Plot2</td>
<td>24</td>
<td>71</td>
<td>181</td>
<td>2,955</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Google Maps</td>
<td>74</td>
<td>834</td>
<td>778</td>
<td>9,080</td>
<td>13</td>
<td>17</td>
<td>16</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>11</td>
<td>0</td>
<td>461</td>
<td>14</td>
</tr>
<tr>
<td>InfoVis</td>
<td>27</td>
<td>231</td>
<td>559</td>
<td>2,340</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>22</td>
<td>7</td>
</tr>
<tr>
<td>jqPlot</td>
<td>82</td>
<td>182</td>
<td>170</td>
<td>1,460</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>jQuery</td>
<td>211</td>
<td>204</td>
<td>35</td>
<td>192</td>
<td>2</td>
<td>4</td>
<td>31</td>
<td>23</td>
<td>7</td>
<td>1</td>
<td>27</td>
<td>26</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>MooTools</td>
<td>50</td>
<td>99</td>
<td>118</td>
<td>528</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>PolyMaps</td>
<td>66</td>
<td>135</td>
<td>547</td>
<td>12,877</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>Three.js</td>
<td>151</td>
<td>1,053</td>
<td>1,749</td>
<td>12,410</td>
<td>100</td>
<td>110</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>442</td>
<td>49</td>
</tr>
<tr>
<td>Visualization</td>
<td>28</td>
<td>87</td>
<td>357</td>
<td>9,454</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>17</td>
<td>2</td>
</tr>
<tr>
<td><a href="http://www.amazon.com">www.amazon.com</a></td>
<td>184</td>
<td>1,038</td>
<td>321</td>
<td>6,477</td>
<td>4</td>
<td>31</td>
<td>17</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>16</td>
<td>15</td>
<td>0</td>
<td>8</td>
<td>6</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td><a href="http://www.apple.com">www.apple.com</a></td>
<td>101</td>
<td>3,666</td>
<td>2,503</td>
<td>34,220</td>
<td>10</td>
<td>19</td>
<td>27</td>
<td>23</td>
<td>4</td>
<td>0</td>
<td>22</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>111</td>
<td>16</td>
</tr>
<tr>
<td><a href="http://www.paypal.com">www.paypal.com</a></td>
<td>73</td>
<td>2,084</td>
<td>2,306</td>
<td>21,261</td>
<td>8</td>
<td>12</td>
<td>19</td>
<td>16</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>37</td>
<td>29</td>
</tr>
<tr>
<td><a href="http://www.microsoft.com">www.microsoft.com</a></td>
<td>134</td>
<td>2,061</td>
<td>1,375</td>
<td>18,121</td>
<td>8</td>
<td>10</td>
<td>20</td>
<td>19</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>100</td>
<td>19</td>
</tr>
<tr>
<td><a href="http://www.youtube.com">www.youtube.com</a></td>
<td>75</td>
<td>2,324</td>
<td>3,196</td>
<td>44,238</td>
<td>8</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>5</td>
<td>0</td>
<td>11</td>
<td>8</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>70</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>1,565</td>
<td>15,789</td>
<td>246</td>
<td>178</td>
<td>54</td>
<td>14</td>
<td>163</td>
<td>132</td>
<td>15</td>
<td>9</td>
<td>132</td>
<td>132</td>
<td>15</td>
<td>36</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Table 3: JSMiner’s pattern mining results |

The pattern involves an if statement and inter-procedural dependencies (between the statements at lines 4 and 6, Figure 11a). The code in Figure 11b is similar; however, the instance is in an assignment (line 1), whereas the instance in Figure 11a is in the field buffer of an object literal (line 2).

5. “Loading page” in www.microsoft.com (Figure 12). This pattern involves the interaction among three fields of an object literal: `nextDelay` for delay time (line 2), `window.loadContents` (line 9), and `onload` to invoke next (line 4). The fields `nextDelay` and `loadContents` are used inside an anonymous function (line 7) and declared outside (lines 2 and 9).

6.5 Anti-pattern Detection

To show a useful application of our representation, we also conducted a study using JSModel to identify JS misuse. First, we collected code examples showing JS programming mistakes (anti-patterns) and their fixes (correct patterns) from various JS guides and forums. We then selected 30 JS applications from Google Code relating to the patterns by using keyword searching. For a system, we ran a subgraph-matching algorithm on the JSModels built for the patterns and those for the JS code of the system to determine if it has any anti-patterns. We were able to identify 11 errors relating to 4 anti-patterns. For example, the JS usage (receiving an AJAX response) is missing a check for the fields `readyState` and `status` and those for the JS code of the system to determine if it has any anti-patterns. We were able to identify 11 errors relating to 4 anti-patterns. For example, the JS usage (receiving an AJAX response) is missing a check for the fields `readyState` and `status` and those for the JS code of the system to determine if it has any anti-patterns. We were able to identify 11 errors relating to 4 anti-patterns.

6.6 Usefulness as Documentation & Templates

Let us outline our controlled experiment to evaluate if patterns mined by JSMiner are useful as documentation and code templates to help in understanding and using JS APIs. We recruited 15 (under)graduate students at Iowa State Uni-
window.MSCOM.Helper.Content = { ...
nextDelay: 100,
logenabled: 11, ...
onload: function () {
    this.log("window.MSCOM.Helper.Content: Body Onload");
    this.isBodyLoaded = 10;
}
setTimeOut($(.proxy(this.next, this), this.nextDelay)
} ... a.type == 'doc' & this.loadDocument(a ... } ...
};

Figure 12: Loading page in www.microsoft.com

1 // http://pyv8.googlecode.com/svn/trunk/demos/env.js
2 var xhr = new XMLHttpRequest(); xhr.open('GET', url);
3 xh.onloadystatechange = function(){
4 // BUG: Missing if (xhr.readyState == 4 &
5 window.document = xhr.responseXML;
6 ...
7 xh.send();

Figure 13: Anti-pattern detection

versity with mixed JS programming experience. Each person was randomly given 9 questions in 3 categories (3 questions in each). (1) In the first category, a correctly mined JS pattern was shown in the skeleton form (e.g., the bold texts in Figure 8b). A participant has 4 options (and a "not sure") to select the description that best describes the shown pattern. Among those four, there is one correct answer (e.g., "sending a request as a user clicks on a loaded page"). This category aims to evaluate the understandability of the mined patterns as documentation expressed in our skeleton form. (2) For 3 questions in the second category, they were asked to rate the usefulness as JS documentation of the mined pattern shown earlier in the first category. We used the following rating scheme: a) very useful ("I will use it as-is, or modify it slightly"), b) somewhat useful ("It is a good starting point, but I will need to make significant modifications"), c) not useful ("I will not use it at all"), and d) not sure. (3) For the third category, we also aim to evaluate usefulness of the mined patterns but as the recommended code templates. We simulated the practice of selecting code templates. We gave a participant a JS programming task and asked him/her to select one among the three mined patterns to complete the task, or indicate "not sure". There is only one correct answer. We manually graded all 135 answers.

Table 4 shows our results. Among 45 questions in category, 31 (69%) were correctly chosen (column Corr/Useful). Only 14 of them were incorrect or "not sure". This result shows that most patterns mined by JSMiner are understandable. Moreover, as seen in the result for category 2, 38 of 45 answers (84.4%) are positive with 24 "very useful" ones, confirming the mined patterns are useful as JS usage documentation. As seen, in category 3, the participants were able to select many correct patterns for the given tasks (27 of 45 answers). The result shows that our mined patterns are useful as code templates for users to perform a task. This also indicates a promising application of the patterns in source code completion for JS programming.

Table 4: Results on usefulness as JS documentation

<table>
<thead>
<tr>
<th>Category</th>
<th>Corr/Useful</th>
<th>Inc/Not Useful</th>
<th>Not Sure</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category 1</td>
<td>31 (69%)</td>
<td>13 (29%)</td>
<td>1 (2%)</td>
<td>45</td>
</tr>
<tr>
<td>Category 2</td>
<td>38 (84%)</td>
<td>6 (13%)</td>
<td>1 (2%)</td>
<td>45</td>
</tr>
<tr>
<td>Category 3</td>
<td>27 (60%)</td>
<td>17 (38%)</td>
<td>1 (2%)</td>
<td>45</td>
</tr>
</tbody>
</table>

7. RELATED WORK

Our prior work, GrouMiner [2], models a usage by a graph called Groum with two types of nodes: action nodes for method calls and field accesses, and control nodes for control structures. In Groum's later version [4], data nodes model the types of variables. JSMiner has several advances. First, GrouMiner models only variables’ types and cannot model nested, aggregated data objects in a pattern, especially with JS functions treated as objects (O2). Second, it is for single-language programs and cannot support cross-language patterns (O3). Third, it cannot work with JS where types might not statically be determined (O5). Finally, its mining algorithm cannot detect inter-procedural patterns (O6).

Chang et al. [5] and Li et al. [6] aim to find patterns and clones on Program Dependence Graph (PDG). Chang et al. use a frequent subgraph mining algorithm to find patterns on condition nodes on PDGs and detect neglected conditions. Li et al. [6] detect cloned buggy code on PDG. Although with PDG, they can address the challenges in O1 (data/ control dependencies in patterns) and O6 (inter-procedural patterns), it cannot overcome the challenges in O2-O5.

Other static pattern mining methods model a usage pattern via a set of pairs, a sequence, or a partial order of method calls [7, 9, 10, 15, 8, 16, 11], a set of entities [14], CTL formulas [17], or a usage template [13, 12]. None of them supports data objects and anonymous functions.

There are dynamic-analysis-based pattern mining methods to detect bugs [26, 27]. They mine execution traces to construct the patterns in terms of FSAs [18, 20], association rules [19], behavioral patterns [21], function precedence [22], probabilistic model [23], and message sequences [19].

8. CONCLUSIONS

This paper addresses a usage pattern mining problem in JS Web applications where JS uses involve unnamed data objects whose types are not statically revealed. We introduce JSMiner, a graph representation for JS uses, and JSMiner, a JS usage pattern miner that mines inter-procedural, data-oriented JS usage patterns with untyped data objects. Our evaluation shows that JSMiner detects JS usage patterns with higher accuracy than a state-of-the-art approach. Our experiments showed JSMiner’s usefulness in detecting anti-patterns and serving as JS documentation and templates.

9. ACKNOWLEDGMENTS

This project is funded in part by US NSF grants: CCF-1018600, CNS-1223828, CCF-1349153, and CCF-1320578.
10. REFERENCES


[29] http://home.engineering.iastate.edu/~hungnv/Research/JScan